CEWES MSRC/PET TR/99-17

Using Fortran 90 Features for Cache Optimization

by

Benjamin Willhoite
Dan Nagle

04h01699

Work funded by the DoD High Performance Computing Modernization Program CEWES Major Shared Resource Center through

Programming Environment and Training (PET)

Supported by Contract Number: DAHC94-96-C0002
Nichols Research Corporation

Using Fortan 90 Features for Cache Optimization

Benjamin Willhoite

Dan Nagle

March 5, 1999

Modern microprocessor-based computers use cache-based memory systems. Historically, much scientific computing has been performed on computers which did not employ caches (other than registers). The main advantage these older systems have over current systems is the large memory-to-processor bandwidth needed to sustain vector operations. Over the years, both processors and memories have gotten much faster. However, the speed of computer memories has not kept pace with the speed of computer processors. Thus, it has gotten harder for memory systems to keep up with the demand that processors may make upon them. This has resulted in manufacturers producing cache-based memory systems.

In a computer system without cache, all memory is "equal"; that is, any word of memory can be accessed as fast as any other. In a computer system with cache, all memory is not equal. Some memory is represented in the cache, and some is not. Memory in the cache can be accessed much faster than memory outside the cache. Caches store the contents of particular locations depending on which locations have been referenced recently during program execution. The memory represented in the cache changes constantly during program execution.

Programs written for non-cache memory systems tend to store data in several arrays. These arrays are processed from beginning to end at each step of the calculation. This style of coding gives very good performance on vector computers. It is, however, a style poorly suited to cache-based computers. Since the arrays are likely to be larger than the cache, processing the whole array will not allow the first elements to stay in cache while the last elements are being processed. So data in the cache are not reused, degrading performance.

This paper discusses cache reuse, illustrating it with several examples, and showing the reader the potential gains and possible pitfalls of attempting to optimize its use. Timings for various examples confirm that different compiler optimizations and coding strategies can significantly affect the results. It should be noted that the algorithms for all of these codes are identical, only the implementations vary from example to example.

Computer Caches

The idea behind the use of caches is that if data are needed once, they will probably be used again shortly. Also, if data from memory are needed, then data from nearby locations in memory will also be needed. This is referred to as locality in time and locality in space. Nearby in memory means that the addresses differ by a small amount. How small is determined by the cache size and varies from machine to machine. Therefore, a cache based processor works best when working repeatedly on relatively small groups of data, all located near to each other in memory. In contrast to vector computers that grandly operate on very wide swathes of data, cache based computers are myopics with magnifying glasses. If they have to look up from what they are doing, they lose time.

Caches are designed to supply the processor at peak speeds. Even so, the latency of a cache is usually one or two clock periods of its processor. The size of a cache times its clock speed divided by its latency give its bandwidth. Note that a processor may have an internal clock rate which applies to operations within the processor and cache, and an external clock rate which is the bus speed to main memory.

It is not unusual for a processor to wait one or two clock periods for a datum in cache. Such a small delay can usually be masked by the compiler or processor by means of instruction scheduling. The delay for access directly to memory may be as high as ten or twenty clock periods. This may well be too long a duration for instruction scheduling to mask effectively.

The next three most important characteristics of a cache are its line size, its set associativity, and its replacement policy. Each will be discussed in turn.

The cache line size is the smallest amount of data that may be moved into the cache as the result of a single memory reference. The value of the line size is usually four, eight or 16 words, or 32, 64 or 128 bytes. If an array begins on an address that is a multiple of the cache line size , say eight words, then when the first word of the array is referenced, words two through eight are also moved into the cache. As a program works its way through this array, only every eighth fetch is to main memory, the rest are to the cache. In general, every (line size)th sequential reference will be to main memory, the rest will be satisfied by the cache.

The cache set associativity is the number of places in cache where a particular word of main memory can be stored. This is usually written as "the cache is n-way set associative," meaning that a particular word of main memory may be stored in any of n storage locations within the cache. Because the main memory is larger than the cache, it will happen that two locations in main memory map to the same location in cache. Thus, having large (say, eight-way or 16-way) set associativity allows more arrays simultaneously to be in cache, even if they would otherwise share the same location.

The replacement policy is how the cache decides which line, in the set associated with a particular word of main memory, will be used to cache the word. The three most popular policies are:

1. *Random Replacement* : the system simply uses a cache line, "without looking"
2. *Least Recently Used* : the cache line not referenced for the longest time is used
3. *Least Frequently Used* : the cache line with the fewest accesses (during some period of time) is used

The best policy is LFU, but it is the most difficult to implement so it is not often used. Most modern caches are LRU, but watch out for random replacement caches. It is possible for a random replacement cache to destroy the performance of a code relying on LRU.

The Example Programs

Table I. Timings in seconds by system and optimization level

Program	Origin 2000				IBM SP			
	FO	**-O3**	**-O2**	**-O0**	**FO**	**-O3**	**-O2**	**-O0**
ONE	0.4	0.3	5.7	11.9	14.7	12.9	12.7	15.0
TWO	0.2	0.2	10.3	42.8	7.2	3.3	3.5	12.3
THREE	11.0	11.0	13.2	20.6	15.1	1.5	1.7	8.2
FOUR	0.4	0.4	0.8	11.2	0.4	0.4	0.6	5.6
FIVE	0.3	0.2	0.8	7.8	0.3	0.4	0.6	7.0

Compiler options can greatly affect the performance of a code. Below are the optimization levels used in this paper, and the compiler options that generate them, from lowest optimization to highest:

1. No optimization. This is the -O0 option on the SGI Origin2000 (O2K). No optimization is the default on the IBM SP (SP).
2. Standard optimization. Generated with -O2 on the O2K and on the SP.
3. Full optimization. Generated with -O3 on the O2K and with -O3 -qstrict on the SP.
4. Aggressive optimization. Generated with -O3 -Ofast=IP27 on the O2K and by -O3 -qhot -qarch=pwr2 on the SP.

All the example programs shown below perform the following operations:

1. B, C, D, and E are initialized with random numbers. This operation is not included in the timing data. For brevity, random initialization is only shown in the first example.
2. Array A is defined from corresponding locations in arrays B, C, D, and E.
3. Arrays C, D, and E are updated from A and B.
4. Array B is updated from C, D, and E.
5. Array A is used to hold the results of the previous operations.

These examples are intended to demonstrate the data access patterns that might be present in modern code. All of the examples are presented as subroutines called by a main program shell. The shell includes definitions, declarations, and timing code, but does not do any work. It is shown in appendix B. The example subroutines, each incrementally restructured to affect compiler optimization, are presented below.

```
subroutine exampleONE(N, a, b, c, d, e, dx, dy, dz)
integer, intent (IN):: N
real, intent (IN):: dx, dy, dz
real, dimension (N, N, N)::  a, b, c, d, e
do k = 1, N
  do j = 1, N
    do i = 1, N
      a(i, j, k) = 0.5 * b(i, j, k) *              &
                       ( c(i, j, k) * c(i, j, k) + &
                         d(i, j, k) * d(i, j, k) + &
```

```
                        e(i, j, k) * e(i, j, k) )
    end do
    do i = i, N
      c(i, j, k) = a(i, j, k) * b(i, j, k) + d(i, j, k)
      d(i, j, k) = a(i, j, k) * d(i, j, k) + b(i, j, k)
      e(i, j, k) = dx * b(i, j, k) + a(i, j, k)
    end do
    do i = 1, N
      b(i, j, k) = c(i, j, k) + d(i, j, k) * e(i, j, k) * 0.5
      a(i, j, k) = dx + dy + dz + a(i, j, k)
    end do
  end do
end do
end subroutine exampleONE
```

The performance of this code is probably good on a vector computer, because each vector is read from beginning to end. There is no conditional code to interrupt the vector pipelines. However, on a cache based computer, the performance of this code is poor, because there is essentially no cache reuse. That is, each time an array element is referenced, there is no copy in cache, so the processor must refer to memory. That is because each array, not to mention all five arrays simultaneously, is too big to fit into cache. Elements already in cache will be flushed to make room for new elements. The program then runs at memory speed rather than the much faster cache speed.

This program optimizes well on the O2K with full optimization. Aggressive optimization gives no additional benefit. Curiously, this code gains almost nothing from optimization on the SP.

Example Two: Array Notation

Most existing Fortran programs use explicit do-loops rather than array notation for array operations. However, many array operations are more easily expressed, and thus understood, when written as array operations. When the programmer is blocking loops for cache reuse, the programmer is rewriting array operations. Using array notation can simplify the rewrite, and result in clearer code. Also, Fortran 90 array syntax implies no order of operations, whereas do-loops specify a particular order of operations. A compiler has greater freedom to reorder operations when expressed in array syntax. As shown in table I, this can greatly affect the success of compiler optimizations.

This version of the code attempts to perform better than the original, by better use of cache. For a given j and k, a column of each array is fetched to cache. Then it is operated on repeatedly, at cache speeds. This strategy allows the cache to work effectively: data is fetched to the cache, used, then reused. Also, nearby words of memory are processed as a group with no intervening reference to other data which may flush the cache.

The previous example, rewritten to use array syntax, is shown below.

```
subroutine ExampleTWO(N, a, b, c, d, e, dx, dy, dz)
integer, intent(IN):: N
real, intent (IN):: dx, dy, dz
real, dimension (N, N, N):: a, b, c, d, e
do k = 1, N
  do j = 1, N
      a(:, j, k) = 0.5 * b(:, j, k) *        &
                   (c(:, j, k) * c(:, j, k) + &
                    d(:, j, k) * d(:, j, k) + &
```

```
                    e(:, j, k) * e(:, j, k))
        c(:, j, k) = a(:, j, k) * b(:, j, k) + d(:, j, k)
        d(:, j, k) = a(:, j, k) * d(:, j, k) + b(:, j, k)
        e(:, j, k) = dx * b(:, j, k) + a(:, j, k)
        b(:, j, k) = c(:, j, k) + d(:, j, k) * e(:, j, k) * 0.5
        a(:, j, k) = dx + dy + dz + a(:, j, k)
    end do
  end do
  end subroutine ExampleTWO
```

Now the only loops that remain explicit are those that are implemented over different blocks. The loops which implemented the operations over the leading dimension are here implied by the array syntax. The compiler is free to implement another level of blocking, perhaps to reblock the operations for another level of cache, for example. The compiler would have to do a sophisticated dependency analysis if the inner loops were explicit. This code shows substantial performance benfits when exposed to even the lowest levels of optimization. The lesson for programmers is clear. If optimizations are left to the compiler, it must be given code that it can clearly understand. Different compilers 'clearly understand' different coding strategies. Table I shows that at lower levels of optimization the O2K compiler provides no speedup. At no optimization it even considerably degrades performance. It is notable that this code looses performance when aggresively optimized on the SP. Programmers should be prepared to test their codes with various levels of optimization.

Example Three

In the examples presented so far, the arrays are declared something like this:

```
integer, parameter:: N = 128
real, dimension (N, N, N)::  a, b, c, d, e
```

Another way of writing the same memory layout is like this:

```
integer, parameter::  N = 128
integer, parameter::  aa=1, bb=2, cc=3, dd=4, ee=5
real, dimension (N, N, N, ee)::  work
```

Now, instead of referencing A(:, :, :), we will reference work(:, :, :, A), and similarly for B, C, D, and E. Note that both of these declarations specify the same memory layout.

The effects of this transformation are shown below:

```
subroutine ExampleTHREE(N, aa, bb, cc, dd, ee, dx, dy, dz, work)
integer, intent (IN)::  N
real, intent (IN):: dx, dy, dz
integer aa, bb, cc, dd, ee
real, dimension (N, N, N, ee)::  work
do k = 1, N
  do j = 1, N
    work(:, j, k, aa) = 0.5 * work(:, j, k, bb) *         &
                        (work(:, j, k, cc) * work(:, j, k, cc) + &
                         work(:, j, k, dd) * work(:, j, k, dd) + &
                         work(:, j, k, ee) * work(:, j, k, ee) )
    work(:, j, k, cc) = work(:, j, k, aa) * work(:, j, k, bb) + &
```

```
                        work(:, j, k, dd)
    work(:, j, k, dd) = work(:, j, k, aa) * work(:, j, k, dd) + &
                        work(:, j, k, bb)
    work(:, j, k, ee) = dx * work(:, j, k, bb) + work(:, j, k, aa)
    work(:, j, k, bb) = work(:, j, k, cc) + work(:, j, k, dd) * &
                        work(:, j, k, ee) * 0.5
    work(:, j, k, aa) = dx + dy + dz + work(:, j, k, aa)
  end do
end do
end subroutine ExampleTHREE
```

Although this code is clear and readable, it has one fatal flaw; The two most rapidly changing indices of the the memory references are at opposite ends of the array expressions. That is, the ':' and the 'AA' in 'work(:, ... ,AA)' are far apart. In order for most compilers to optimize array notation, the innermost dimensions must change most rapidly. This allows each array reference to draw in its cache line neighbors with a single reference, allowing cache reuse. This is not the case in this code. It is interesting to note that the SP compiler was not fooled by this scheme, and managed good performance except at very aggressive optimization. The O2K compiler could not optimize this code.

Example Four

We will now rewrite the previous example another way to enable cache issues to be seen more clearly. Previously, we had:

```
integer, parameter::  N = 128
integer, parameter::  aa=1, bb=2, cc=3, dd=4, ee=5
real, dimension (N, N, N, ee)::  work
```

Now we will write:

```
integer, parameter:: N = 128
integer, parameter:: aa=1, bb=2, cc=3, dd=4, ee=5
real, dimension (ee, N, N, N)::work
```

With this rewrite, the program will use work(AA, :, :, :) in place of work(:, :, :, AA), and similarly for the other arrays. Now, what has happened is that each point (I, J, K) in space has all data values (all five 'planes of data') located in consecutive storage locations. This is coded by moving the most rapidly changing indices to the innermost position, and corrects the optimization problem inherent to the last example. The rewritten code appears below:

```
subroutine ExampleFOUR(N, aa, bb, cc, dd, ee, dx, dy, dz, work)
integer, intent (IN)::  N
real, intent (IN):: dx, dy, dz
integer aa, bb, cc, dd, ee
real, dimension (ee, N, N, N):: work
do k = 1, N
  do j = 1, N
    work(aa, :, j, k) = 0.5 * work(bb, :, j, k) *                &
                        ( work(cc, :, j, k) * work(cc, :, j, k) + &
                          work(dd, :, j, k) * work(dd, :, j, k) + &
                          work(ee, :, j, k) * work(ee, :, j, k) )
    work(aa, :, j, k) = work(bb, :, j, k) * work(cc, :, j, k) + &
                        work(dd, :, j, k)
```

```
    work(ee, :, j, k) = dx * work(ee, :, j, k) + work(aa, :, j, k)
    work(bb, :, j, k) = work(cc, :, j, k) + work(dd, :, j, k) * &
                        work(ee, :, j, k) * 0.5
    work(aa, :, j, k) = work(aa, :, j, k) + dx + dy + dz
  end do
end do
end subroutine ExampleFOUR
```

This example now has all data values for a given spatial point stored in consecutive locations. Thus, referencing work(A, ...) has the effect of also moving work(B, ...), work(C, ...), work(D, ...) and work(E, ...) into the cache. Another way of describing the layout is to say that the storage locations are 'interleaved'. Note that although the speedup is good, it is also available at much lower levels of optimization because the compiler can understand better where it will be looking for data next.

Example Five

One further basic optimization may be possible. Many compilers compute the offset from the array base using the array indices at run time. A new feature with Fortran 90 is the user defined Derived Type, similar to the C struct.

To rewrite our example using derived types, we proceed as follows:

```
type::  work_t
  real a, b, c, d, e
end type
integer, parameter:: N = 128
type(work_t), dimension (N, N, N)::work
```

Now, instead of referencing A(I, J, K), we reference work(I, J, K)%A. The following example illustrates:

```
subroutine ExampleFIVE(N, dx, dy, dz, work)
integer, intent (IN)::  N
real, intent (IN):: dx, dy, dz
type::  work_t
  real a, b, c, d, e
end type
type(work_t), dimension (N, N, N)::work
do k = 1, N
  do j = 1, N
    work(:, j, k)%a = 0.5 * work(:, j, k)%b *           &
                     (work(:, j, k)%c * work(:, j, k)%c + &
                      work(:, j, k)%d * work(:, j, k)%d + &
                      work(:, j, k)%e * work(:, j, k)%e)

    work(:, j, k)%c = work(:, j, k)%a * work(:, j, k)%b + &
                      work(:, j, k)%d
    work(:, j, k)%d = work(:, j, k)%a * work(:, j, k)%d + &
                      work(:, j, k)%b
    work(:, j, k)%e = dx * work(:, j, k)%b + work(:, j, k)%a
    work(:, j, k)%b = work(:, j, k)%c + work(:, j, k)%d + &
                      work(:, j, k)%e * 0.5
    work(:, j, k)%a = dx + dy + dz + work(:, j, k)%a
  end do
end do
```

```
      end subroutine ExampleFIVE
```

The advantage of this code is that it is very understandable to the compiler. When any data point is referenced, it pulls all of its neighbors with it. Finally, it is marginally easier to code, because all five array planes are combined in a single symbol. This scheme achieves speedup comparable to any of the other codes.

Conclusion

Cache-based memory systems require a substantially different memory layout than traditional vector computers for optimal performance. Reworking the memory layout can be accomplished by rewriting loops. Modifying code in this manner is tedious and error-prone. However, the use of Fortran 90 features such as array syntax and derived types can result in a more reliable transition.

The new microprocessor-based computers must be programmed with an eye towards their memory bandwidths if their impressive peak computational speeds are to be obtained. Blocking older codes for cache reuse, while tedious, is a necessary part of making the transition successful.

Appendix A: Effects of Caches in Parallel Computation

When porting code to a shared memory multiprocessor computer, there are several caches. (In a distributed memory computer, one also has several caches. However, the issue of cache consistency never comes up because all inter-memory, and hence all inter-cache, communication is typically via message passing.)

This leads to two phenomena: False Sharing and Superluminal Speedup (also called Superlinear Speedup). Each will be discussed in turn.

If two different processors attempt to use the same word in memory, each will fetch it to their respective caches. For reasons of shared memory consistency, if a word in cache is to be updated, its value in memory must be updated as well. Then copies of the word in other caches must be invalidated, and the "fresh" value obtained from memory. Thus, if two processors both attempt to update the same word in memory, both will, in turn, invalidate the cache copy of the other. This forces both processors to use the memory copy, and run at memory speed rather than cache speed. Informally, the processors are said to be "thrashing each other’s caches." However, since both processors are using the same memory copy, both processors are operating on the same data values, and so should return consistent results regardless of the exact order of the updates (which, of course, is why the cache copies are invalidated).

Recall that it is not a word that is cached, but rather a whole cache line consisting of, say, eight words. Thus, it is not a word that must be invalidated, but rather a whole cache line. Suppose A(I) and A(I+1) map to the same cache line. Suppose further that one processor is updating A(I), while another processor is updating A(I+1). Then, when the first processor invalidates all other processors’ cache copy of A(I), it will also invalidate all other processors’ cache copy of A(I+1). Likewise, when the second processor updates A(I+1), it will invalidate all other processors copy of A(I) as well. This situation is called false sharing.

The result of false sharing is this: If two different processors are to work on different data, typically different array elements of the same array, the different processors must work on different cache lines. One result of this is that the division of data between the two processors must occur on a cache line

boundary. In that way, both processors can simply operate out their respective caches, without interfering with the other.

Superluminal Speedups occur when, as the number of processors in a parallel computation of fixed size increases and the amount of data per processor decreases, the amount of data per processor becomes smaller than the cache size of a single processor. Thus, the entire working set of a processor fits into the processor's cache, and the programmer observes that the execution rate of the whole program increases more than would be indicated by the increase in the number of processors alone.

Appendix B: Example code outer shell

```
      program test
      implicit none
!
!     Appropriate definitions for example in question
      integer, parameter :: N = 128

      integer i, j, k, start, finish, rate
      real dx, dy, dz, seconds

!     Appropriate initialization for example in question, usually
      call random_number(a)
      call random_number(b)
      call random_number(c)
      call random_number(d)
      call random_number(e)

      call System_Clock (COUNT = start)

      dx = 0.333
      dy = 0.667
      dz = 0.125

!     Call subroutine for example in question, varies by example
      call sub (N, a, b, c, d, e, dx, dy, dz)

      call System_Clock (COUNT = finish, COUNT_RATE = rate)
      seconds = real (finish - start) / real (rate)
      print *,'Time taken:', seconds, ' seconds.'

      end program test
```